

# Matthew Duc Nguyen

dnguy521@ucr.edu

University of California, Riverside

## Parallel Quicksort

### 1 Introduction

This report explains the implementation of parallel quicksort on an array of  $10^9$  numbers. Through the implementation, from an average of 5 trials, sorts array in 9.87 seconds from sampling and finding the median of the first 1000 elements as the pivot.

Code: <https://github.com/ucparlay-class/project1-matthewduc>

### 2 Quicksort

Through multiple testing of the best granularity control, sorting an array of with less than  $3.5^7$  elements is faster with `std::sort`.

#### 2.1 Partition Function

```
template <typename T>
size_t Filter(T* A, size_t n,
              size_t control, size_t pivot,
              size_t* output, size_t position,
              size_t* flag, size_t* ps){
size_t* LS = new size_t[n-1];
cilk_for(int i=0;i<n;i++){
    flag[i] = 0;
    ps[i] = 0;
// Less Than Pivot
    if (control == -1){
        if (A[i] < pivot){
            flag[i] = 1;
        }
    }

// Equal To Pivot
    if (control == 0){
        if (A[i] == pivot){
            flag[i] = 1;
        }
    }
}
```

```
    }
}

// Greater Than Pivot
    if (control == 1){
        if (A[i] > pivot){
            flag[i] = 1;
        }
    }
}

std::atomic<size_t> increment = position;
cilk_for (size_t i = 0; i<n;i++){
    if(flag[i] == 1){
        output[increment++] = A[i];
    }
}

position = increment;
return position;
}
```

This function takes an array pointer, array size, a control value, a pivot, and pointer to an output array, a position value, and a pointer to a flag array as the parameters. Given the pivot and control, the flag array will flag elements less than, equal to, and greater than the pivot. The flags and control will copy the elements into its correct partition in the output array and return the pointer.

#### 2.2 Quicksort Function

```
template <typename T>
void quicksort(T* A, size_t n) {
    size_t leftPointer, middlePointer,
           rightPointer;
    if (n <= 35000000){
        std::sort(A, A+n);
        return;
    }
}
```

```

} else {
// Sample to find pivot, e.g. 1000
size_t* B = new size_t[1000];
cilk_for(int i=0; i<1000; i++){
    B[i] = A[i];
}
// Sort B and use middle number as pivot
std::sort(B, B+1000);
size_t pivot = B[500];
// Partition into Left, Equal, Right
size_t* output = new size_t[n];
size_t* flagLess = new size_t[n];
size_t* equalTo = new size_t[n];
size_t* flagGreater = new size_t[n];
size_t* ps = new size_t[n];

leftPointer = Filter(A, n, -1, pivot,
output, 0, flagLess, ps);
middlePointer = Filter(A, n, 0, pivot,
output, leftPointer, equalTo, ps);
rightPointer = Filter(A, n, 1, pivot,
output, middlePointer, flagGreater, ps);

cilk_for(int i=0; i<n; i++){
    A[i] = output[i];
}
}
cilk_spawn quicksort(A, leftPointer);
quicksort(A+middlePointer, n-middlePointer);
cilk_sync;
}

```

This function initializes a leftPointer, middlePointer, and rightPointer. The leftPointer will point to the last position in the partition whose elements are less than the pivot. To handle repetitions, the middlePointer will point to the last position in the partition whose elements are equal to the pivot. The pivot is chosen from the median of 1000 elements in the partition, this will increase the chances of picking a good pivot to call the partition.

### 2.3 Parallel Sort

```

cilk_spawn quicksort(A, leftPointer);
quicksort(A+middlePointer, n-middlePointer);
cilk_sync;

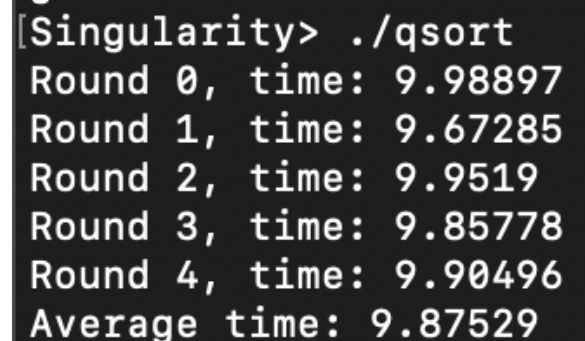
```

After retrieving the pointers to the end of the left partition and the beginning of the right partition, the two partitions may be sorted in parallel processes.

## 3 Granularity and Runtime

Selecting the best granularity control began with using if partition  $n \leq 50000000$  and 1000 samples to find the pivot, the average run time from 5 rounds is 12.54 seconds. Using  $n \leq 100000000$  and 1000 samples to find the pivot, produced an average run time of 13.03 seconds. With  $n \leq 350000000$  and 1000 samples to find the pivot, the average run time is 9.87 seconds. With the best partition size of  $n \leq 350000000$ , using 10, 100, 10000 samples from the partition and using the middle number as the pivot produced 10.41 seconds, 10.47 seconds, and 10.30 seconds average run times

The best control is found to be  $n \leq 350000000$  and using 1000 elements from the partition to find the middle pivot. From 5 round of running the parallel quicksort algorithm, the average run time is 9.87 sec.



```

[Singularity> ./qsort
Round 0, time: 9.98897
Round 1, time: 9.67285
Round 2, time: 9.9519
Round 3, time: 9.85778
Round 4, time: 9.90496
Average time: 9.87529

```

## 4 Conclusion

Quicksort can run in parallel to partition the array based on the pivot and point to the start positions of each partition. If the partition size is small enough, found in this project to be  $n \leq 350000000$ , `std::sort()` can be invoked to sort the remaining parts of the large array. From the testing of different number of samples in the partition to find a good pivot, the middle number from 1000 samples was found to be the best pivot to complete the partitions. The best average runtime results in 9.87 seconds.

## 5 Acknowledgment

CS214 has definitely solidified my knowledge in C++ programming and taught me a new field in Algorithms, to run things in parallel. Thank you Professor Yiham for a very enlightening quarter.